Folding a binary operator: a neglected technique

0. We consider a binary operator $\oplus$ of type $X \times A \to X$, for some fixed types X and A. In what follows dummies b and c have type A whereas x has type X. With $\oplus$ we can form expressions, of type X, like:

$$\begin{array}{c} x \\ x \oplus b \\ (x \oplus b) \oplus c \ . \end{array}$$

What these expressions have in common is, except their type, that they depend on one value of type X and some —zero or more— values of type A. Because the order of these values is relevant — $(x\oplus b)\oplus c$ is not the same as $(x \oplus c) \oplus b$ — they can conveniently be considered as the elements of a list of type $\mathcal{L}(A)$. Thus, the expressions can be viewed as functions of an X and an $\mathcal{L}(A)$; we now make this explicit by introducing a new binary operator $\otimes$, of type $X \times \mathcal{L}(A) \to X$, which we require to satisfy:

$$\begin{array}{rcl} x &=& x \otimes [] \\ x \oplus b &=& x \otimes [b] \\ (x \oplus b) \oplus c &=& x \otimes [b,c] \ . \end{array}$$

Moreover, because the expression $(x\oplus b) \oplus c$ is an instance of the expression $x \oplus b$, namely with $x,b := x \oplus b, c$, our new operator $\otimes$ must, for the sake of consistency, also satisfy:

$$x \otimes [b,c] \ = \ (x \oplus b) \otimes [c] \ .$$

By simple generalisation — replacing [c] by any list — we obtain the following recursive definition for $\otimes$ (in which s has type $\mathcal{L}(A)$):

$$
\begin{array}{lrcl}
(0) & x \otimes [\,] & = & x \\
(1) & x \otimes ([b] +\!\!+ s) & = & (x \oplus b) \otimes s
\end{array}
$$

Apart from (0) and (1), $\otimes$ has other interesting properties:

$$
\begin{array}{lrcl}
(2) & x \otimes [b] & = & x \oplus b \\
(3) & x \otimes (t +\!\!+ s) & = & (x \otimes t) \otimes s \\
(4) & x \otimes (t +\!\!+ [c]) & = & (x \otimes t) \oplus c
\end{array}
$$

((2) follows from (1) and (0), (3) is proved from (0) and (1) by induction on t, and (4) follows from (3) and (2).)

As a recursive definition (0) and (4) serve equally well as (0) and (1); this flexibility provides a basis for program transformations: whenever we encounter the one definition we may replace it by the other if we like. Rule (1), for example, is tail recursive whereas rule (4) is not.

Some properties of $\oplus$ are inherited by $\otimes$. For example, if $\oplus$ distributes from the right over some binary operator $\odot$ then so does $\otimes$:

<u>lemma:</u> if $(\oplus b)$ distributes over $\odot$
then $(\otimes s)$ distributes over $\odot$

<u>proof</u>: by induction on s.
$\square$

Our operator $\otimes$ is well-known; it is called foldl ($\oplus$) in R.S. Bird and Ph. Wadler 's book "Introduction to functional programming" (Prentice-Hall) 1988). Only recently I became aware that its properties represent often occurring patterns in functional programming and that, therefore, it pays to know these properties. Whenever I use the technique I shall introduce $\otimes$ as "the (left) folded version of $\oplus$".

Of course, the technique has a mirror image: for operators of type $A \times X \to X$ we have (right) folded versions of type $\mathcal{L}(A) \times X \to X$, with mirror image properties.

Notice that the above development is completely devoid of operational interpretations: we have introduced $\otimes$ and deduced its properties by looking at the shapes of a few formulae only and by distilling what they have in common.

1. We apply the above to the following example. We consider function F, of type $\mathcal{L}(A) \to X$, which is defined in terms of a constant E and a binary operator $\oplus$, as follows:

$$\begin{aligned} F.[] &= E \\ F.([b] \mathbin{+\!\!+} s) &= b \oplus F.s \end{aligned}$$

Apparently, $\oplus$ has type $A \times X \to X$ and we have:

$$\begin{aligned} F.s &= [] \otimes F.s \\ F.([b] \mathbin{+\!\!+} s) &= [b] \otimes F.s \quad , \end{aligned}$$

from which we may get the idea that F could satisfy the more general property, for all t, s:

(5) $F.(t +\!\!+ s) \;=\; t \otimes F.s$.

This property holds indeed, as the following proof by induction on t shows:

(i) $F.([\,] +\!\!+ s)$
$=$ { $[\,]$ identity of $+\!\!+$ }
$F.s$
$=$ { definition of $\otimes$ }
$[\,] \otimes F.s$,

and:
(ii) $F.([b] +\!\!+ t +\!\!+ s)$
$=$ { definition of F (using $+\!\!+$'s associativity) }
$b \oplus F.(t +\!\!+ s)$
$=$ { Ind. Hyp. }
$b \oplus (t \otimes F.s)$
$=$ { definition of $\otimes$ }
$([b] +\!\!+ t) \otimes F.s$.

By instantiating (5) with $s := [\,]$ and using $F.[\,] = E$ we obtain:

(6) $F.t \;=\; t \otimes E$.

Because neither the expression $t \otimes E$ nor the definitions of $\otimes$ contain F, formula (6) can be used as a definition of F. Using one pair of properties of $\otimes$ we obtain:

$$\begin{aligned} [\,] \otimes x \;&=\; x \\ ([b] +\!\!+ t) \otimes x \;&=\; b \oplus (t \otimes x) \end{aligned}$$,

which is just F's original definition in which constant E

has been replaced by a parameter. Using another pair, however, we obtain a tail recursive definition:

$$\begin{aligned} [\,] \otimes x &= x \\ (t \mathbin{+\!\!+} [b]) \otimes x &= t \otimes (b \oplus x) \end{aligned}$$

A minor complication is that, as exhibited by the patterns $t \mathbin{+\!\!+} [b]$ and $[b] \mathbin{+\!\!+} s$ (in F's definition), the elements of the list are interpreted in reversed order. If so desired this can be remedied by introduction of a new operator $\bar{\otimes}$ with specification:

$$t \mathbin{\bar{\otimes}} x = rev.t \otimes x \, .$$

This is a standard transformation that leads to the following solution:

$$\begin{aligned} F.(rev.t) &= t \mathbin{\bar{\otimes}} E \quad , \quad \text{where} \\ [\,] \mathbin{\bar{\otimes}} x &= x \\ ([b] \mathbin{+\!\!+} t) \mathbin{\bar{\otimes}} x &= t \mathbin{\bar{\otimes}} (b \oplus x) \, . \end{aligned}$$

This is the Tail Recursion Theorem for (finite) lists.

2. Substitution is a ternary operation mapping two expressions and a name onto an expression. By grouping one expression and the name into a pair we can consider substitution as a binary operator as well, of type $Exp \times (Name \times Exp) \to Exp$. Here we denote this operator by $\triangleleft$; with s and t of type Exp and v of type Name we write

$$s \triangleleft \langle v, t \rangle$$

instead of the more usual $s(v := t)$ or (uglier) $s(v \backslash t)$.

We now denote the folded version of $\lhd$ by $\text{\textcircled{$\lhd$}}$; by means of $\text{\textcircled{$\lhd$}}$ we can study lists of substitutions, and because $\text{\textcircled{$\lhd$}}$ is the folded version of $\lhd$ we obtain some of its properties for free. With C and D for lists of (name, expression) pairs we have for example:

$$\begin{aligned}
s \text{ \textcircled{$\lhd$} } (C \mathbin{+\!\!+} D) &= (s \text{ \textcircled{$\lhd$} } C) \text{ \textcircled{$\lhd$} } D \\
s \text{ \textcircled{$\lhd$} } ([\langle v,t\rangle] \mathbin{+\!\!+} D) &= (s \lhd \langle v,t\rangle) \text{ \textcircled{$\lhd$} } D \\
s \text{ \textcircled{$\lhd$} } (C \mathbin{+\!\!+} [\langle v,t\rangle]) &= (s \text{ \textcircled{$\lhd$} } C) \lhd \langle v,t\rangle \\
(s \odot t) \text{ \textcircled{$\lhd$} } C &= (s \text{ \textcircled{$\lhd$} } C) \odot (t \text{ \textcircled{$\lhd$} } C)
\end{aligned}$$

The latter property holds for every expression constructor $\odot$ because substitution distributes over the expression constructors. We shall see applications of these properties when we discuss evaluators for λ-calculus expressions.

3. As a final example, we consider the following recursive definition of the binary operator $\oplus$ —explanation follows— :

$$\begin{aligned}
x \oplus b \;=\; &\underline{\text{if}}\; B_0 \to y_0 \\
&\;[]\; B_1 \to y_1 \oplus c_1 \\
&\;[]\; B_2 \to (y_2 \oplus c_2) \oplus d_2 \\
&\underline{\text{fi}} \;,
\end{aligned}$$

in which the B_i denote boolean expressions in terms of x and b and in which y_i, c_i, and d_i denote arbitrary expressions in terms of x and b, with y_i of type X and c_i and d_i of type A.

The double recursion in the third alternative of this definition can be eliminated as follows. We introduce the folded version of $\oplus$, called $\otimes$ again, inspired by the shape of the expression $(y_2 \oplus c_2) \oplus d_2$. We observe that:

$$x \oplus b \quad = \quad x \otimes [b] \ ,$$

which can be used as a definition of $\oplus$ if only we are able to derive a definition of $\otimes$ in which $\oplus$ does not occur anymore. For the most complicated case this derivation runs as follows:

```
    $x \otimes ([b] +\!\!+ s)$
=       { definition of $\otimes$ }
    $(x \oplus b) \otimes s$
=       { definition of $\oplus$, case $B_2$ }
    $((y_2 \oplus c_2) \oplus d_2) \otimes s$
=       { definition of $\otimes$ (twice) }
    $y_2 \otimes ([c_2, d_2] +\!\!+ s)$ .
```

In this way the following definition for $\otimes$ is obtained:

```
$x \otimes []$             =  $x$
$x \otimes ([b] +\!\!+ s)$  =  if $B_0 \rightarrow y_0 \otimes s$
                           ▯ $B_1 \rightarrow y_1 \otimes ([c_1] +\!\!+ s)$
                           ▯ $B_2 \rightarrow y_2 \otimes ([c_2, d_2] +\!\!+ s)$
                           fi
```

Again, this derivation is entirely devoid of operational connotations; although the list parameter may be interpreted as the stack needed for the implementation of recursion, this interpretation has played no role in the derivation.

Eindhoven, 2 september 1994
Rob R. Hoogerwoord